08_intro_to_neural_networks

May 10, 2024

This portfolio contains information from Matt William's course Introduction to Neural Networks.

1 Introduction to neural networks

Neural networks are a type of machine learning model that are inspired by the human brain. Artificial neurons are a simplification of biological neurons – they take in multiple inputs and pass on their output to multiple other neurons. A neuron's value is determined by the sum of its inputs (x_i), which are weighted by the strength of the connection between the neurons (w_i):

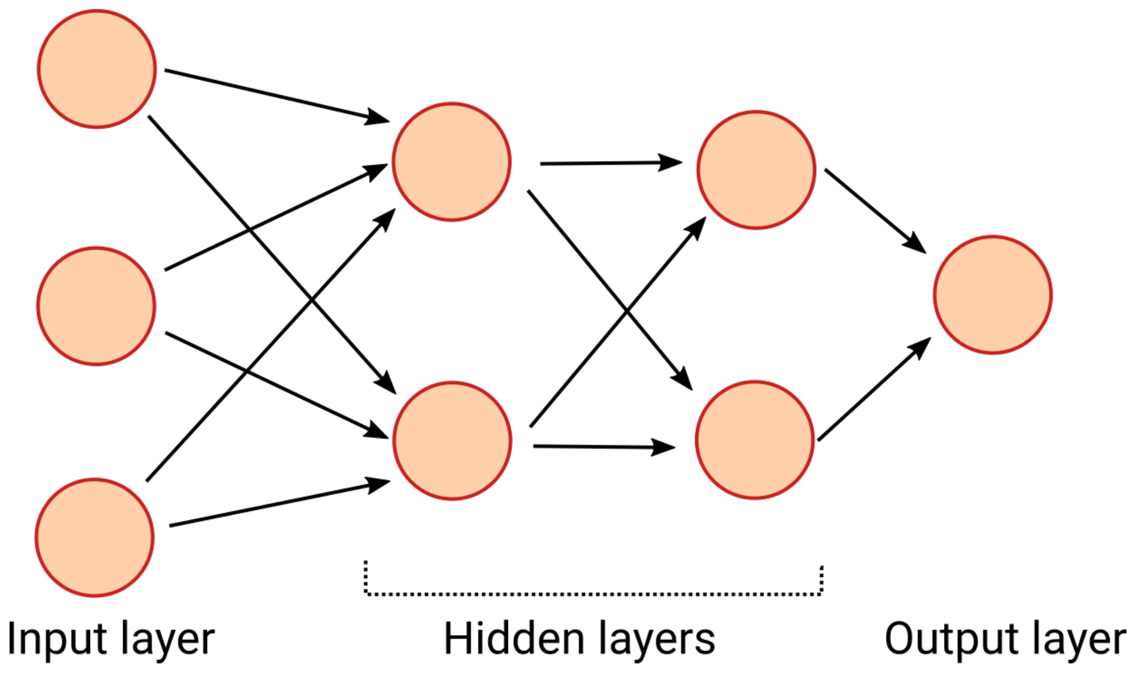
$$p = \sum_i x_i w_i$$

This value p is then passed through an activation function ϕ to determine the neuron's output o:

$$o = \phi(p) = \phi\left(\sum_i x_i w_i\right)$$

This activation function step allows neural networks to model non-linear relationships.

Most neural networks have the following architecture:



The input layer typically takes on values from the user / data. There are then a number of hidden layers, which are made up of neurons that take in the input layer's values and pass on their output to the next layer. The output layer then takes in the output of the last hidden layer and produces the final output of the network.

Deciding the shape of a neural network is not an exact science. Typically, the size of the hidden layers is similar to the size of the input and output layers. The number of hidden layers represents the level of abstraction the network can learn.

2 Training a neural network

Neural networks are most commonly trained using backpropagation. Initial weights can be set randomly. We define a loss function that measures how far off the network's output is from the true value. We then calculate the derivative of the loss function with respect to the weights and adjust the weights accordingly.

3 Python environments

Python environments are a copy of Python plus the packages you need. You can create a new environment using Anaconda Navigator by clicking on the `Environments` tab and then `Create`. You can then search for and select the packages you need. For this portfolio, we need `tensorflow`, `tensorflow-datasets`, `scikit-learn`, `scikit-image`, `jupyterlab`, and `pandas`.

4 Convolutional neural networks

Convolutional neural networks (CNNs) are a class of neural networks primarily used for image recognition and processing tasks. They leverage image kernel convolution by applying series of learned filters across input images. Through layers of of filters, CNNs can efficiently extract relevant features, e.g. edges, enabling them to recognize patterns and objects within images.

CNNs typically contain three types of layers:

- Convolutional layers: consist of a set of convolution filters, also known as kernels, to the image. The convolution operation involves sliding the filter over the input image, computing the element-wise product between the filter and the corresponding patch of the input, and summing the results to generate a single value in the output feature map.
- Pooling layers: downsample the resulting image data, resulting in decreased dimensionality and thus lower computing time and lower risk of overfitting. A common method of pooling is known as max pooling, where the maximum value within a fixed window is retained while discarding the rest.
- Dense (fully connected) layers: perform classification on the features extracted after convolution and pooling. These are traditional neural network layers where every neuron is connected to every neuron in the preceding layer.

5 MNIST example

We will show an example application using CNNs on the MNIST dataset, which is a dataset consisting of 70,000 28×28 pixel images of handwritten digits.

5.1 Designing the CNN

1. First convolutional layer: consists of 16 5×5 filters (layer size = 28×28×16=12544)
2. First pooling layer: reduces the image by a factor of 2 in all directions (layer size = 14×14×16=3136)
3. Second convolutional layer: consists of 32 5×5 filters (layer size = 14×14×32=6272)
4. Second pooling layer: reduces the image by a factor of 2 in all directions (layer size = 7×7×32=1568)
5. Dense layer: fully-connected layer of 128 nodes
6. Output layer: 10 neurons corresponding to the 10 classes (digits from 0-9)

[1]:
```python
import tensorflow as tf

model = tf.keras.models.Sequential([
    # first convolutional layer
    tf.keras.layers.Conv2D( # layer for 2-dimensional image
        filters=16,
        kernel_size=5,
        padding="same", # what to put on the edges
        activation=tf.nn.relu
    ),
    # first pooling layer
    tf.keras.layers.MaxPool2D((2, 2), (2, 2), padding="same"),
    # second convolutional layer
    tf.keras.layers.Conv2D(
        filters=32,
        kernel_size=5,
        padding="same",
        activation=tf.nn.relu
    ),
    # second pooling layer
    tf.keras.layers.MaxPool2D((2, 2), (2, 2), padding="same"),
    # flatten layer into a linear set of nodes
    tf.keras.layers.Flatten(),
    # add a fully connected layer of 128 nodes
    tf.keras.layers.Dense(128, activation="relu"),
    # use drop-out regularization to randomly ignore 40% of the nodes each␣
  ↪training cycle
    tf.keras.layers.Dropout(0.4),
    # output layer
    tf.keras.layers.Dense(10, activation="softmax")
])
```

```
2024-05-10 11:29:29.767248: I tensorflow/core/util/port.cc:113] oneDNN custom
operations are on. You may see slightly different numerical results due to
floating-point round-off errors from different computation orders. To turn them
off, set the environment variable `TF_ENABLE_ONEDNN_OPTS=0`.
2024-05-10 11:29:29.830014: E
```

```
external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:9261] Unable to register
cuDNN factory: Attempting to register factory for plugin cuDNN when one has
already been registered
2024-05-10 11:29:29.830054: E
external/local_xla/xla/stream_executor/cuda/cuda_fft.cc:607] Unable to register
cuFFT factory: Attempting to register factory for plugin cuFFT when one has
already been registered
2024-05-10 11:29:29.831377: E
external/local_xla/xla/stream_executor/cuda/cuda_blas.cc:1515] Unable to
register cuBLAS factory: Attempting to register factory for plugin cuBLAS when
one has already been registered
2024-05-10 11:29:29.841945: I tensorflow/core/platform/cpu_feature_guard.cc:182]
This TensorFlow binary is optimized to use available CPU instructions in
performance-critical operations.
To enable the following instructions: SSE4.1 SSE4.2 AVX AVX2 AVX_VNNI FMA, in
other operations, rebuild TensorFlow with the appropriate compiler flags.
2024-05-10 11:29:32.340975: I
external/local_xla/xla/stream_executor/cuda/cuda_executor.cc:901] successful
NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
2024-05-10 11:29:32.391204: I
external/local_xla/xla/stream_executor/cuda/cuda_executor.cc:901] successful
NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
2024-05-10 11:29:32.391469: I
external/local_xla/xla/stream_executor/cuda/cuda_executor.cc:901] successful
NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
2024-05-10 11:29:32.395012: I
external/local_xla/xla/stream_executor/cuda/cuda_executor.cc:901] successful
NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
2024-05-10 11:29:32.395249: I
external/local_xla/xla/stream_executor/cuda/cuda_executor.cc:901] successful
NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
2024-05-10 11:29:32.395410: I
external/local_xla/xla/stream_executor/cuda/cuda_executor.cc:901] successful
```

```
NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
2024-05-10 11:29:32.485899: I
external/local_xla/xla/stream_executor/cuda/cuda_executor.cc:901] successful
NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
2024-05-10 11:29:32.486152: I
external/local_xla/xla/stream_executor/cuda/cuda_executor.cc:901] successful
NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
2024-05-10 11:29:32.486337: I
external/local_xla/xla/stream_executor/cuda/cuda_executor.cc:901] successful
NUMA node read from SysFS had negative value (-1), but there must be at least
one NUMA node, so returning NUMA node zero. See more at
https://github.com/torvalds/linux/blob/v6.0/Documentation/ABI/testing/sysfs-bus-
pci#L344-L355
2024-05-10 11:29:32.486477: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1929] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 1583 MB memory:  -> device: 0,
name: NVIDIA GeForce MX550, pci bus id: 0000:02:00.0, compute capability: 7.5
```

5.2 Training the model

We will train the model using sparse categorical cross-entropy for the loss function. This loss function is well-suited to situations where we have more than two classes.

[2]:
```python
model.compile(
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"],
)
```

Next, we load the data and split it into training and test sets. The resulting training and test sets are sequences of 28×28×1 matrices containing the numbers 0-255, each with a label from 0-9 indicating which number is in the image:

[3]:
```python
import tensorflow_datasets as tfds

ds_train_orig, ds_test_orig = tfds.load(
    "mnist",
    split=["train", "test"],
    as_supervised=True,
)
```

```
/home/aw23877/anaconda3/envs/nn/lib/python3.11/site-packages/tqdm/auto.py:21:
TqdmWarning: IProgress not found. Please update jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

We need to convert our data from the range 0-255 to 0-1. Then we can shuffle the data and put it into batches of 128.

[4]:
```
def normalize_img(image, label):
    return tf.cast(image, tf.float32) / 255., label

ds_train = ds_train_orig.map(normalize_img)
ds_train = ds_train.shuffle(1000).batch(128)

ds_test = ds_test_orig.map(normalize_img)
ds_test = ds_test.batch(128)
```

Now we can fit the model to the data:

[5]:
```
model.fit(
    ds_train,
    validation_data=ds_test,
    epochs=2,
)
```

```
Epoch 1/2
```

```
2024-05-10 11:29:34.258062: I
external/local_xla/xla/stream_executor/cuda/cuda_dnn.cc:454] Loaded cuDNN
version 8907
2024-05-10 11:29:35.538022: I external/local_xla/xla/service/service.cc:168] XLA
service 0x7e6ae8765b20 initialized for platform CUDA (this does not guarantee
that XLA will be used). Devices:
2024-05-10 11:29:35.538082: I external/local_xla/xla/service/service.cc:176]
StreamExecutor device (0): NVIDIA GeForce MX550, Compute Capability 7.5
2024-05-10 11:29:35.546907: I
tensorflow/compiler/mlir/tensorflow/utils/dump_mlir_util.cc:269] disabling MLIR
crash reproducer, set env var `MLIR_CRASH_REPRODUCER_DIRECTORY` to enable.
WARNING: All log messages before absl::InitializeLog() is called are written to
STDERR
I0000 00:00:1715336975.629805   85734 device_compiler.h:186] Compiled cluster
using XLA!  This line is logged at most once for the lifetime of the process.
```

```
469/469 [==============================] - 8s 11ms/step - loss: 0.2560 -
accuracy: 0.9211 - val_loss: 0.0611 - val_accuracy: 0.9792
Epoch 2/2
469/469 [==============================] - 3s 7ms/step - loss: 0.0742 -
accuracy: 0.9780 - val_loss: 0.0391 - val_accuracy: 0.9871
```

[5]:
```
<keras.src.callbacks.History at 0x7e6c0c4b1490>
```

From the output of `fit`, we can see the following values for each epoch:

- loss: The value of the loss function calculated on the training data for the last batch in the epoch
- accuracy: The fraction of the entries in the training data set that are classified correctly
- val_loss: The value of the loss function calculated on the test data
- val_accuracy: The fraction of the entries in the test set that are classified correctly

5.3 Testing on real-world images

We can now test the model on some additional real-world images. We have 10 images of 28x28 pixels apiece, with 1 color channel (black and white).

[6]:
```
from urllib.request import urlretrieve

for i in list(range(1,10)) + ["dog"]:
    urlretrieve(f"https://github.com/milliams/intro_deep_learning/raw/master/
 ↪{i}.png", f"{i}.png")

import numpy as np
from skimage.io import imread

images = []
for i in list(range(1,10)) + ["dog"]:
    images.append(np.array(imread(f"{i}.png")/255.0, dtype="float32"))
images = np.array(images)[:,:,:,np.newaxis]
images.shape
```

[6]:
```
(10, 28, 28, 1)
```

We can apply the model to these images to make predictions.

[7]:
```
probabilities = model.predict(images)

truths = list(range(1, 10)) + ["dog"]

table = []
for truth, probs in zip(truths, probabilities):
    prediction = probs.argmax()
    if truth == 'dog':
        print(f"{truth}. CNN thinks it's a {prediction} ({probs[prediction]*100:
 ↪.1f}%)")
    else:
        print(f"{truth} at {probs[truth]*100:4.1f}%. CNN thinks it's a␣
 ↪{prediction} ({probs[prediction]*100:4.1f}%)")
    table.append((truth, probs))
```

```
1/1 [==============================] - 0s 302ms/step
1 at  3.9%. CNN thinks it's a 3 (43.6%)
```

```
2 at 86.5%. CNN thinks it's a 2 (86.5%)
3 at 30.3%. CNN thinks it's a 8 (48.6%)
4 at  1.4%. CNN thinks it's a 3 (94.4%)
5 at 100.0%. CNN thinks it's a 5 (100.0%)
6 at  3.2%. CNN thinks it's a 5 (44.7%)
7 at 55.5%. CNN thinks it's a 7 (55.5%)
8 at 24.5%. CNN thinks it's a 3 (46.1%)
9 at  1.6%. CNN thinks it's a 8 (78.0%)
dog. CNN thinks it's a 2 (37.0%)
```

5.4 Data augmentation

We can see some issues in performance when we apply our model to real-world images that might not have the exact same image composition as our training data, which contained only white handwritten numbers on a black background. To address this, we can try some data augmentation strategies to create a wider range of input images. We will do so by adding color-inverted images to our training dataset and then retraining the model:

[8]:
```python
# Function for inverting color of the image
def invert_img(image, label):
    return 1.-image, label

ds_train_new = ds_train_orig.map(normalize_img)
ds_train_new = ds_train_new.concatenate(ds_train_new.map(invert_img))  # add␣
 ↪inverted images to training
ds_train_new = ds_train_new.shuffle(1000)
ds_train_new = ds_train_new.batch(128)

ds_test_new = ds_test_orig.map(normalize_img)
ds_test_new = ds_test_new.concatenate(ds_test_new.map(invert_img))  # add␣
 ↪inverted images to testing
ds_test_new = ds_test_new.batch(128)

# Retrain the model:
model.fit(
    ds_train_new,
    validation_data=ds_test_new,
    epochs=2,
)
```

```
Epoch 1/2
938/938 [==============================] - 7s 8ms/step - loss: 0.0938 -
accuracy: 0.9714 - val_loss: 0.1039 - val_accuracy: 0.9653
Epoch 2/2
938/938 [==============================] - 7s 7ms/step - loss: 0.0587 -
accuracy: 0.9820 - val_loss: 0.0605 - val_accuracy: 0.9800
```

[8]:
```
<keras.src.callbacks.History at 0x7e6c0c289490>
```

We can now check to see if there is any improvement in the model's performance on the additional real-world images:

```
[9]: probabilities = model.predict(images)

     truths = list(range(1, 10)) + ["dog"]

     table = []
     for truth, probs in zip(truths, probabilities):
         prediction = probs.argmax()
         if truth == 'dog':
             print(f"{truth}. CNN thinks it's a {prediction} ({probs[prediction]*100:
      ↪.1f}%)")
         else:
             print(f"{truth} at {probs[truth]*100:4.1f}%. CNN thinks it's a␣
      ↪{prediction} ({probs[prediction]*100:4.1f}%)")
         table.append((truth, probs))
```

```
1/1 [==============================] - 0s 32ms/step
1 at 11.9%. CNN thinks it's a 8 (18.4%)
2 at 100.0%. CNN thinks it's a 2 (100.0%)
3 at 100.0%. CNN thinks it's a 3 (100.0%)
4 at 99.6%. CNN thinks it's a 4 (99.6%)
5 at 98.5%. CNN thinks it's a 5 (98.5%)
6 at 100.0%. CNN thinks it's a 6 (100.0%)
7 at 98.7%. CNN thinks it's a 7 (98.7%)
8 at 99.9%. CNN thinks it's a 8 (99.9%)
9 at 18.6%. CNN thinks it's a 8 (41.4%)
dog. CNN thinks it's a 8 (22.6%)
```

We can see that the model still has trouble with the 1 and 9, but the model now correctly identifies the 2-4, 6, and 8.